ICS 354 – Automata and Language Translation Systems
Lecture Notes on the Theory of Computation (LNTC)

Introduction

Each branch of Computer Sciences has its own objects of study, and it is time we turned more specifically to those of the theory of computation. The *theory of computation* is the mathematical study of computing machines and their capabilities. We must therefore develop a model for the data that computers manipulate. This is a big area in the theoretical computer science. Unfortunately, we are not going to discuss lots of interesting problems and theorems here since they are out of the scope of this course. However, we will just touch few issues related to the area of modeling computation to get the feeling of studying this topic. We adopt the mathematically expedient choice of representing data by strings of symbols. The best way to start this topic is by introducing the notion of alphabet and languages.

The format of the lecture notes is as follows. All concepts and issues (like definitions, theorems, notations, and special notes) are sequentially numbered in brackets [like these] so that the student may need to study and understand them in sequence. Some exercises are given at the end of each section to ensure understanding of the material. Each section has a reference to the corresponding sections in the textbook (marked with §)

1. Alphabet and Languages (§1.2)

There are many languages in real life. A language consists of words. Each word consists of letters which are taken from a set of symbols called the alphabet. For Example, the alphabet of English consists of 26 symbols (letters). A sequence of letters forms a string that can be a word in the language. For example, the strings *apple* and *book* are two words in English, but the string *aaabbbccc* is not, even though they are strings on the same alphabet. The binary language (or code) is defined on the alphabet {0, 1}. For any given machine language, some strings on {0, 1} are recognized as words (or valid input) and some are not.

In this section, we will use the same analogy to define languages in order to study the computing machines and their capabilities. Let us start a formal approach by introducing the following definitions:

[1] *alphabet*: a finite set of symbols, denoted by Σ

Example:	$\Sigma = \{a, b, c\}$	// has 3 symbols
	$\Sigma = \{a, <, 3, \#, @, 6\}$	// any object can be a symbol
	$\Sigma = \{0, 1\}$	// has 2 symbols, for binary strings
	$\Sigma = \{a, b, c, \ldots, z\}$	// 26 symbols, for English
	$\Sigma = \varnothing$	// could be empty; has zero symbols

[2] A *string on* Σ: is a finite sequence of *zero or more* symbols from Σ

Example: If $\Sigma = \{a, b, c\}$, then *abbbc* is a string on Σ, and *aa* is another string.
If $\Sigma = \{0, 1\}$, then 001 and all binary codes are strings on Σ

[3] The *length* of a string s, denoted by $|s|$, is the number of symbols in *s*.
Example: The length of *abbbc* is 5, and the length of 0001 is 4.
$|a| = 1$
If $s = aaa$, then $|s| = 3$

[4] The *empty string* (or *null*): is a string of length zero. i.e. it has no symbols. The empty string is defined over every alphabet and it is denoted by λ (Lambda). Therefore, by definition, $|\lambda| = 0$

[5] **Note:** Although any object can be a symbol, we usually use common characters like letter and numerals as symbols. The alphabet {a, b} is frequently used instead of the binary alphabet. Thus, the two most used alphabets are {a, b} & {0,1}. If Σ is not given explicitly and cannot be determined from the context, then $\Sigma = \{a, b\}$ by default.

[6] *concatenation* of strings: let *x* and *y* be two strings on the same alphabet. The concatenation of *x* and *y* forms a third string denoted by $x \cdot y$ or simply: *xy*
Example: Let x = aaab and y = ba, then xy = aaabba
Note: for any string x over some Σ, $x \cdot \lambda = x = \lambda.x$

[7] **Notation:** s^n denotes the *repeated concatenation* of *s*, which is the string obtained by concatenating *s* to itself *n* times. So, $s^2 = s \cdot s$ and $s^3 = s \cdot s \cdot s$. In other words, s^n can be define recursively as follows:

$$s^n = s^{n-1} \cdot s \text{ and } s^0 = \lambda$$

Similarly, for any symbol *a* in some alphabet, a^n denotes the string made of the symbol *a* repeated *n* times.
Example: Let $s = abba$, then $s^2 =$ abbaabba, and $s^3 =$ abbaabbaabba.
The string $a^4 = aaaa$, and $b^3 = bbb$.

[8] In general, the *repeated concatenation* can be applied on any set of symbols, A, and it means the set of all strings made of *n* symbols from A.
Thus, $A^n = \{s \mid s$ is a string of length n over A$\}$
Example: Σ^n is the set of all strings of length *n* over Σ

[9] *Kleene star* operation: For any symbol $a \in \Sigma$, a^* is the set of all strings made of the symbol *a* repeated *zero or more* times.
Thus, $a^* = \{\lambda, a, aa, aaa, aaaa, \ldots\} = \{s \mid s = a^n \text{ for all } n \geq 0\}$

[10] In general, the *Kleene start* operator can be applied on any set of symbols, A, and it means the set of all strings made of zero or more symbols from A.
Thus, $A^* = \{s \mid s$ is a string on A$\}$
Example: Σ^* is the set of all strings on Σ.
$\{a, c\}^* = \{\lambda$, a, c, aa, ac, ca, cc, aaa, aac, aca, caa, acc, ccc, aaaa, … $\}$

[11] *Substring*: a string y is a *substring* of w if and only if there are two strings $x, z \in \Sigma^*$ such that $w = xyz$. (Notice that x and/or z can be null). If x is null, then the substring y is called a *prefix* of w; and if z is null, then y is a *suffix* of w.
Example: *aa* and *bb* are two substrings of *abbaa*, *aa* is also a suffix. λ is a substring of every string.

[12] The *reverse* of a string s, denoted by s^R, is the same string spelled backwards. For example, $(abac)^R$ = caba. **Note**: the upper case R is reserved for the reversal operator. (Do not confuse with the notation of repeated concatenation s^n in [7])

[13] **Theorem 1.1:** For any two strings $x, y \in \Sigma^*$, $(x \cdot y)^R = y^R \cdot x^R$

[14] A *language*, *L*, is a set of strings on an alphabet Σ. Thus, $L \subseteq \Sigma^*$
Example: L_1 = {a, aaa, abba} is a language defined over Σ = {a, b}
L_2 = {aaa} is another language over the same alphabet
$L_3 = a^*$ is also a language over {a, b}, not using b though.
$L_4 = \Sigma^*$, a language that has all strings.
$L_5 = \varnothing$ is the empty language

[15] *Word*: the strings in a language are also called *words*
Example: In the above example, L_1 has 3 words and L_2 has one word.
The word aaa is a common word in both languages, L_1 and L_2
L_3 has infinitely many words, including aaa.
L_5 has zero words.

[16] The *string-function*. string w can be considered as a function defined as follows:
w:{1, 2, …, |w|}$\rightarrow \Sigma$, where w(i) is the i^{th} symbol in w.

[17] The *language of palindromes*: $L_p = \{w \mid w = w^R\}$ is the language of all palindrome words, such as: aba, aa, abbbba, a, b and λ. L_p is infinite, i.e. it has infinitely many words.

[18] **Note**: most languages of interest are infinite, so that listing all the strings is not possible. Thus we can define infinite languages, as we define sets in general, by the following schemes:

1. L = $\{w \in \Sigma^* \mid w$ has some property$\}$ (set definition)
2. L = $L_1 \cup L_2$ (union of two languages)
3. L = $L_1 \cap L_2$ (intersection of two languages)
4. and by string operations, like: concatenation and Kleene star. See [18] and [19]

We will study more ways to define infinite languages in the following sections.

[19] *Language concatenation*: $L_1 \cdot L_2 = \{w \mid w = x \cdot y$ where $x \in L_1$ and $y \in L_2\}$
Notation: L^n denotes the set of all strings obtained by concatenating n strings from L.
Example: {ab, bb}·{a, b, λ} = {aba, abb, ab, bba, bbb, bb}
$\{ab, b\}^3$ = {ababab, ababb, abbab, babab, abbb, babb, bbab, bbb}

[20] *Closure operation* (or Kleene star): L^* is the set of all strings obtained by concatenating *zero or more* strings from L. (The concatenation of *zero* strings is λ)
Example: $\{ab, bb\}^* = \{\lambda, ab, bb, abab, abbb, bbbb, bbab, ababab, \ldots\}$

[21] **Notation:** L^+ is the set of all strings obtained by concatenating *one or more* strings from L. Thus, $L^+ = L \cdot L^*$. (Note: $\lambda \notin L^+$ unless $\lambda \in L$)

[22] **Theorem 1.2:** For any two languages A and B, if $A \subseteq B$, then $A^* \subseteq B^*$

Example 1.1: Let $A = \{a, ab, b, abbba\}$ be a language defined over $\Sigma = \{a,b\}$. Find A^*.
Solution: Notice that $\Sigma = \{a, b\} \subseteq A$. Therefore $\Sigma^* = \{a, b\}^* \subseteq A^*$ (by Theorem 1.2)
But Σ^* is the set of all strings on Σ. Therefore, $A^* \subseteq \Sigma^*$, and hence $A^* = \Sigma^*$

Exercises on Section 1:

1. Let $\Sigma = \{0, 1\}$, which of the followings are strings on Σ
    a. 0
    b. 110
    c. 012
    d. 0^n
    e. $0^n \cdot 1^n$ for some $n \geq 1$
    f. 0101010101…. (endless string of alternating 0s and 1s)
    g. λ

2. Let A and B be two arbitrary languages on $\Sigma = \{a, b\}$. True or False:
    a. $A \subseteq \Sigma$
    b. $A^* = \Sigma^*$
    c. $(A \cup B)^* \subseteq \Sigma^*$
    d. $A^* \cup B^* = (A \cup B)^*$
    e. $A^* \cap B^* = (A \cap B)^*$

3. True or false? And justify your answer.
    a. $0^* \cup 1^* = \{0, 1\}^*$
    b. $0^* \cap 1^* = \varnothing$

4. Prove by mathematical induction that $(x^R)^n = (x^n)^R$ for all $n \geq 1$.

5. Let $\Sigma = \{a, b\}$. For each of the following languages on Σ, find, if possible, two strings $x, y \in \Sigma^*$ such that x is a word in the given language, and y is not.
    a. $\{\lambda, a, b, ab, ba, aa, bb, aaa, aab, aba, abb, baa, bab, bba, bbb\}$
    b. $\{ab, ba, aa, bb\}^*$
    c. $\{a, bb, ba\}^*$
    d. $(a^* \cdot b^*) \cup (b^* \cdot a^*)$

6. Let $\Sigma = \varnothing$. How many different languages can be defined on Σ?

2. Regular Expressions and Language Representation (§3.1)

A central issue in the theory of computation is the representation of languages by finite specifications. Naturally, any finite language is amenable to finite representation by exhaustive enumeration of all the strings in the language. The issue becomes challenging only when infinite languages are considered.

Let us be more precise about the notion of "finite representation of a language." The first point to be made is that any such representation must itself be a string, a finite sequence of symbols over some alphabet. Second, we certainly want different languages to have different representations, otherwise the term representation could hardly be considered appropriate. We will see in the following sections that these two requirements already imply that the possibilities for finite representation are severely limited.

The first representation method we will study is called *regular expression* (RE). Since any language, L, by definition is nothing but a (possibly infinite) set of strings on some Σ, we would like to know which strings are there in L and which are not. We need precise expressions to define languages besides the ordinary set definitions and operations. So, let us start by a formal definition of regular expressions, and then we see some examples of how such expressions can be used to define languages.

[1] A *regular expression* (RE) over an alphabet Σ is a string on the alphabet Δ, where
$\Delta = \{*, +, (,), \varnothing\} \cup \Sigma$
Thus, Δ contains all the symbols in Σ plus the following symbols: *, +, (,), and $\varnothing$ such that the following rules hold:

1. the elements in Σ and $\varnothing$ are the basic regular expressions
2. if α is a regular expression, then so is α^* (star)
3. if α and β are regular expressions, then so is $\alpha\beta$ (concatenation)
4. if α and β are regular expressions, then so is $\alpha+\beta$ (union)

Note: nothing else is regular expression unless it follows from rules 1 through 4 with some parentheses if needed to control the precedence.

Example:
Let $\Sigma = \{a, b\}$, then we have 3 basic REs (by rule 1), each represents a language, namely:

$\alpha_1 = a$ which represents the language $\{a\}$,
$\alpha_2 = b$ which represents the language $\{b\}$, and
$\alpha_3 = \varnothing$ which represents the language $\varnothing$

By rule 2, we can build three more REs:

a^*	- for the language $\{\lambda, a, aa, aaa, aaaa, \ldots\}$
b^*	- for the language $\{\lambda, b, bb, bbb, bbbb, \ldots\}$
$\varnothing^*$	- for the language $\{\lambda\}$

By rule 3, we can build more REs, like:

aa	- for {aa}
$(aa)^*$	- for all strings of even number of a's (and no b's)
$a^*(b^*)$	- for $\{\lambda, a, b, ab, aab, aabb, abb, abbb, aaaabbb\ldots\}$ and so on.

By combining rules 2, 3 and 4, we can build REs for more complicated languages, like:

(a+b)*	- for Σ^*
a(a+b)*bb	- all strings start with 'a' and end with 'bb'

[2] Precedence rules: the * comes first, then concatenation, and then + comes third.
Example:

ab*a = a(b*)a
a + b a = a + (ba)
(a + b*)a* = (a + (b*)) (a*)
b*(ab*ab*)* = (b*)(a(b*)a(b*))*

[3] **Notation:** in rule 4, the '+' means 'or' and some books use '|' or '$\cup$' instead.

[4] **Notation:** if α is a RE, then $L(\alpha)$ denotes the language represented by α. Since every RE represent a language, we can say that two REs α and β are equivalent (denoted by $\alpha \equiv \beta$) if and only if they represent the same language. Thus, $\alpha \equiv \beta$ *iff* $L(\alpha) = L(\beta)$. For simplicity, we may use the equal sign '=' for equivalency and the language of REs. For example, we may write: $\alpha = \beta = L(\beta)$ when we talk about the set represented by the RE.

Example 2.1: Which of the following regular expressions represents a language that contains the word aa?

ab*a
a + b a
(a + b) a
b*(ab*ab*)*

Solution:

aa $\in$ L(ab*a)	because it is a(b*)a and b* can be λ
aa $\notin$ L(a + ba)	because concatenation has higher precedence.
aa $\in$ L((a + b)a)	because of the parentheses
aa $\in$ L(b*(ab*ab*)*)	because it is the set of all strings with even number of a's.

[5] *Regular languages*: L is *regular* if it can be represented by a regular expression. Thus, L is regular if there is a regular expression α such that $L = L(\alpha)$.
Example: L = {aa, ab, bb, ba} is regular for α = aa + ab + bb + ba
L = {*w* | *w* has no a} is regular for α = b*
L = {*w* | *w* = $a^n b^n$ for $n \geq 1$ } is not regular. (See Exercise 4)
L_p (the language of palindromes) is not regular.

[6] **Theorem 2.1:** Regular languages are closed under union, intersection, complement, concatenation, and star operations.

[7] **Theorem 2.2:** Any finite language is regular.

Exercises on Section 2:

1. Let $\Sigma = \{a, b\}$. For each of the following languages, determine if it is a regular or not, and give a regular expression that represent each of the regular ones.
    a. the set of all strings on Σ that end with a.
    b. the set of all strings on Σ that end with a or b.
    c. the set of all strings on Σ that start with a or end with b.
    d. the set of all strings on Σ of even length.
    e. the set of all strings on Σ that contain the substring bbb.
    f. the set of all strings on Σ that do not contain the substring bbb

2. Let $\Sigma = \{a, b\}$. For each of the following regular languages over Σ, find two strings $x, y \in \Sigma^*$ such that x is a word in the given language, and y is not. (If there is no such x or y, write “none”)
    a. a + ab
    b. (ab* + b)*
    c. (a*b*)*
    d. a* (ba + b + aa)*

3. Prove Theorem 2.2.

4. Argue why a*b* does not represent $L = \{w \mid w = a^n b^n \text{ for } n \geq 1\}$.

3. Finite-State Automata (§2.1)

Finite state automata (or finite automata for short) are good models for computers with an extremely limited amount of memory. What can a computer do with such a small memory?

Many useful things!

In fact, we interact with such computers all the time, as they lie at the heart of various electromechanical devices. Finite automata and their probabilistic counterpart chains are useful tools when we are attempting to recognize patterns in data. These devices are used in speech processing and in optical character recognition. We will now take a close look at finite automata from a mathematical perspective.

We will develop a precise definition of a finite automaton, terminology for describing and manipulating finite automata, and theoretical results that describe their power and limitations.

Besides giving us a clearer understanding of what finite automata are and what they can and cannot do, the theoretical development allows us to practice and become more comfortable with mathematical definitions, theorems, and proofs in a relatively simple setting.

[1] **State Diagrams:**
In beginning to describe the mathematical theory of finite automata, we do so in the abstract, without reference to any particular application. The following figure depicts a finite automaton called Ml.

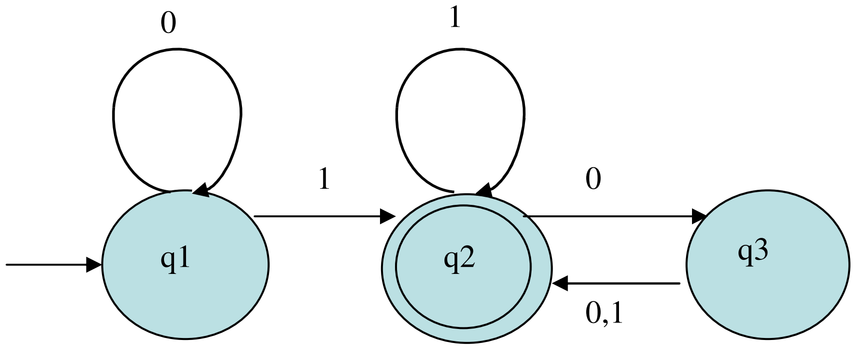

Figure 1: A finite automaton called M1 that has three states

Figure 1 is called the *state diagram* of M1. It has three states, labeled ql, q2, and q3. The *start state*, q1, is indicated by the arrow pointing at it from nowhere. The *accept state*, q2 is the one with a double circle. The arrows going from one state to another are called *transitions*. Each transition is labeled with an input symbol.

When this automaton receives an input string such as 1101, it processes that string and produces an output. The output is either *accept* or *reject*. We will consider only this yes/no type of output to keep things simple.

The processing begins in Ml's start state. The automaton receives the symbols from the input string one by one from left to right. After reading each symbol, Ml moves from one state to another along the transition that has that symbol as its label. When it reads the last symbol, M1 produces its output. The output is *accept* if M1 is now in an accept state and *reject* if it is not.

Example 3.1: Does M1 accept the string 1101?

Solution:
When we feed the input string 1101 to the machine M1 in Figure 1, the processing proceeds as follows.

1. start in state q1;
2. read 1, follow transition from q1 to q2;
3. read 1, follow transition from q2 to q2;
4. read 0, follow transition from q2 to q3;
5. read 1, follow transition from q3 to q2;
6. accept because M1 is in an accept state q2 at the end of the input.

Example 3.2: Test M1 on the following input strings: 1, 01, 11, 0101010101, 100, 0100, 110000, 0101000000, 0, 10, and 101000.

Solution:
Experimenting with this machine on a variety of input strings reveals that it accepts the strings 1, 01, 11, and 0101010101. In fact, Ml accepts any string that ends with 1, as it goes to its accept state q2 whenever it reads the symbol 1. In addition, it accepts strings 100, 0100, 110000, and 0101000000, and any string that ends with an even number of 0s following the last 1. It rejects other strings, such as 0, 10, 101000. Can you describe the language consisting of all strings that M1 accepts? (Exercise 2.a)

[2] *Deterministic* machines: In the above example, M1 is *deterministic*. This means that at every state, q, when the machine M1 reads an input symbol, it knows (or determines) where to go next according to the input symbol and the current state. We call this type of machines Deterministic Finite Automata (DFA). We will see another type of machines (in Section 4) where the next state is not precisely determined. Hence, they are called Nondeterministic Finite Automata (NFA).

Now we define deterministic finite automata formally. Though state diagrams are easier to grasp intuitively, we need the formal definition, too, for two specific reasons:

First, a formal definition is precise. It resolves any uncertainties about what is allowed in a finite automaton. If you were uncertain about whether finite automata were allowed to have 0 accept states or whether they must have exactly one transition exiting every state for each possible input symbol, you could consult the formal definition and verify that the answer is yes in both cases.

Second, a formal definition provides notation. Good notation helps you think and express your thoughts clearly.

[3] A *deterministic finite automaton* (DFA) is a quintuple (Q, Σ, δ, s, F), where

1. Q is a finite set called the *states*,
2. Σ is a finite set called the *alphabet*,
3. $\delta: Q \times \Sigma \rightarrow Q$ is the *transition function*,
4. $s \in Q$ is the *start state*, and
5. $F \subseteq Q$ is the set of *accept states* (also called *final states*).

The formal definition precisely describes what we mean by a finite automaton. For example, returning to the earlier question of whether 0 accept states is allowable, you can see that setting F to be the empty set Ø yields 0 accept states, which is allowable. Furthermore, the transition function δ specifies exactly one next state for each possible combination of a state and an input symbol. That answers our other question affirmatively, showing that exactly one transition arrow exits every state for each possible input symbol.

Example 3.3: Describe M1 formally.

Solution:
M1 = (Q, Σ, δ, q1, F) where

1. Q = {ql, q2, q3},
2. Σ = {0, 1},
3. δ is the transaction table described as follows:

δ	0	1
ql	ql	q2
q2	q3	q2
q3	q2	q2

4. q1 is the start state, and
5. F = {q2}

[4] The l*anguage of a machine* M, denoted by L(M), is a set, A, of all strings accepted by M. We say that M recognizes the language A if L(M) = A.

Example 3.4: Design a DFA that recognizes the language
L = {w | w does not contain the substring bbb}

Solution:
The idea is to accept after 0, 1 or 2 consecutive b's but reject if you ever reach three consecutive b's. Any other combination of a's and b's is accepted. We can draw the DFA, M2, as shown in Figure 2.

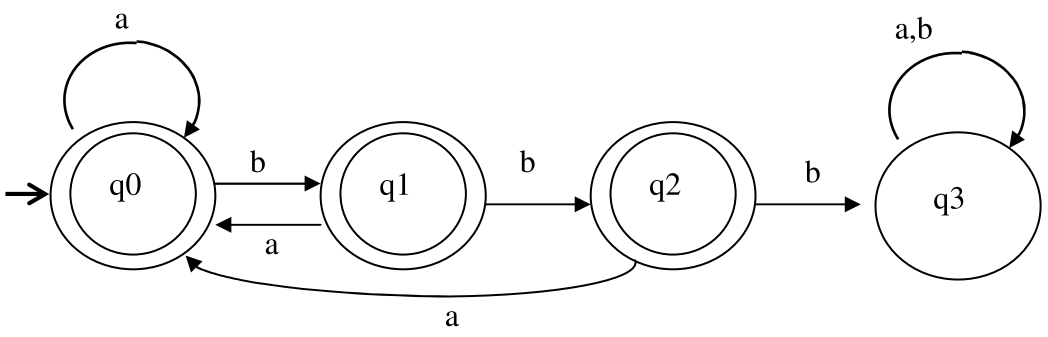

Figure 2: DFA M2 for {w | w does not contain the substring bbb}

M2 = (Q, Σ, δ, q0, {q0, q1, q2}), with Q = {q0, q1, q2, q3}, Σ = {a, b}, and the transition function δ given in the table below, does indeed accepts the specified language.

δ	a	b
q0	q0	q1
q1	q0	q2
q2	q0	q3
q3	q3	q3

[5] The *extended transition function*: When the transition function δ is applied on a state q1 $\in$ Q and a symbol a $\in$ Σ, it yields the next state q2 $\in$ Q. This is denoted by δ(q1, a) = q2. For example, in the DFA M2, δ(q1, a) = q0, and δ(q1, b) = q2. It is convenient to introduce the *extended transition function* δ^*: $Q \times \Sigma^* \to Q$. This function takes an initial state and a string, and it outputs the state that the automaton will be in after reading the string. For example, δ^*(q2, aaab) = q1 in M2.

[6] **Theorem 3.1:** The languages recognized by DFA are regular. Thus, every DFA has an equivalent regular expression that represents the same language.

[7] **Theorem 3.2:** Every regular expression has an equivalent DFA. Thus, if α is a RE, then there is a DFA, M, such that L(M) = L(α).

Exercises on Section 3:

1. Let Σ = {a, b}. Draw a DFA that recognizes:
    a. Σ^*
    b. $\varnothing$
    c. {λ}
    d. all strings of even length
    e. all strings that start with a and end with bb
    f. ab*
    g. all strings that contain the substring aba
    h. all strings that do not contain the substring aba
    i. all strings that have even a's and even b's

2. Write a regular expression for the language of the following DFA:
    a. M1 (shown in Figure 1)
    b. M2 (shown in Figure 2)

4. Nondeterministic Finite Automata (§2.2)

In this section we add a powerful feature to finite automata. This feature is called *nondeterminism*, and is essentially the ability to change states in a way that is only partially determined by the current state and input symbol. That is, we shall now permit several possible "next states" for a given combination of current state and input symbol.

The automaton, as it reads the input string, may choose at each step to go into any one of these legal next states; the choice is not determined by anything in our model, and is therefore said to be nondeterministic.

On the other hand, the choice is not wholly unlimited either; only those next states that are legal from a given state with a given input symbol can be chosen.

A nondeterministic finite automaton (NFA) can be a much more convenient device to design than a deterministic finite automaton (DFA). We will show some examples.

[1] Two main differences between DFA and NFA:

1. NFA have multiple next states on the same input (zero or more next states)
2. NFA allow λ-transitions

Example 4.1: (multiple next states)
This example explains the first difference. Consider the language L = (ab+aba)*, which is accepted by the deterministic finite automaton illustrated in Figure 3.

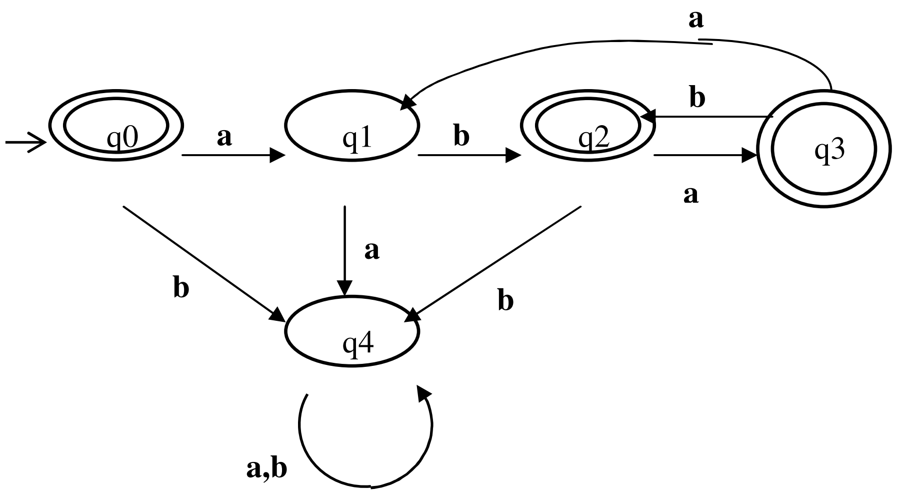

Figure 3: DFA for L = (ab + aba)*

Even with the diagram, it takes a few moments to ascertain that a deterministic finite automaton is shown; one must check that there are exactly two arrows leaving each node, one labeled a and one labeled b. Some thought is needed to convince oneself that the language accepted by this fairly complex device is the simple language L = (ab + aba)*.

However, L is accepted by the simple nondeterministic device shown in Figure 4.

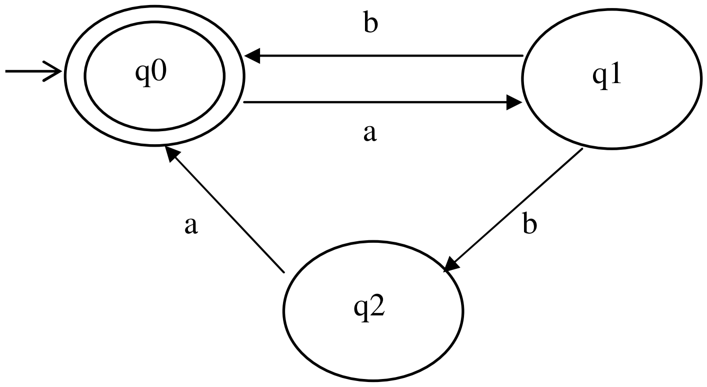

Figure 4: NFA for L = (ab + aba)*

When this device is in state q1, and the input symbol is b, there are two possible next states, q0 and q2. Thus Figure 4 does not represent a deterministic finite automaton. Nevertheless, there is a natural way to interpret the diagram as a device accepting L. A string is accepted if there is some way to get from the initial state (q0) to a final state (in this case, q0) while following arrows labeled with the symbols of the string. For example, ab is accepted by going from q0 to ql to q0; aba is accepted by going from q0 to q1 to q2 to q0.

Of course, the device might guess wrong and go from q0 to q1 to q0 to q1 on input aba, winding up in a non-final state, but this does not matter, since there is *some way* of getting from the initial to a final state with this input.

On the other hand, the input abb is not accepted, since there is no way to get from q0 back to q0 while reading this string. Indeed, you will notice that from q0 there is no state to be entered when the input is b.

Example 4.2: (λ-transition)
Sometimes it is convenient to allow state diagrams in which arrows can be labeled either by symbols in Σ or by the empty string λ. We illustrate λ-transitions in this example.

Consider the device of Figure 5. This device accepts the same language L = (ab + aba)* as in Example 4.1 above. From q2 this machine can return to q0 either by reading an a or immediately, without consuming any input.

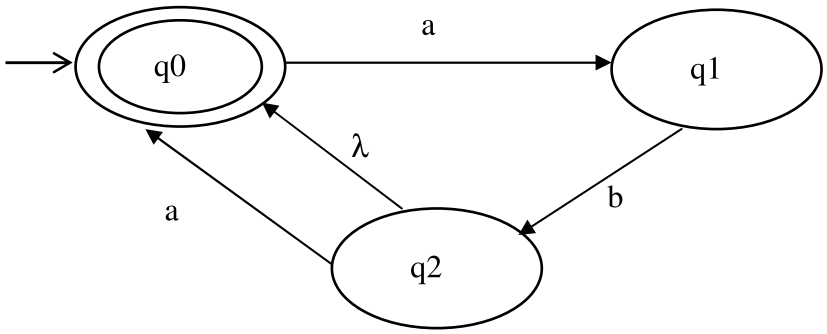

Figure 5: NFA with λ-transition for L = (ab + aba)*

[2] A *nondeterministic finite automata* (NFA) is a quintuple M = (Q, Σ, Δ, s, F), where
1. Q is a finite set of states,
2. Σ is a finite alphabet,
3. δ: Q x (Σ ∪ {λ}) → 2^Q is the *transition function.* (2^Q is the power-set of Q)
4. s ∈ Q is the *start state*, and
5. F ⊆ Q is the set of *final states*.

Note: δ here can be described as a table of the function that maps the current state and an input symbol or λ to a subset of states belongs to the power-set of Q.

Example 4.3: Describe the NFA M3, given in Figure 6, formally.

Solution:
M3 can be defined formally as follows:
M3 = (Q, Σ, Δ, q0, {q2}), where Q = {q0, q1, q2}, and Σ = {0, 1}. The table of the transitions Δ can be described as:

Δ	0	1	2	λ
q0	{q0}	Ø	Ø	{q1}
q1	Ø	{q1}	Ø	{q2}
q2	Ø	Ø	{q2}	Ø

This gives a precise definition of the NFA and the language it recognize. For example, the word 002 belongs to L(M3) because it is accepted by M3 through the path q0, q0, q0, q1, q2, q2 with arcs labeled 0, 0, λ, λ, 2.

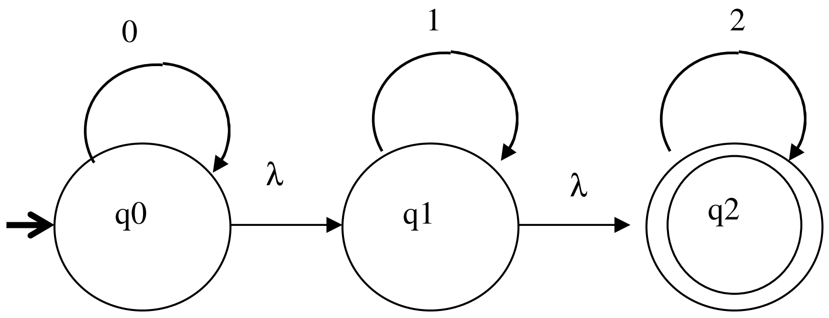

Figure 6: M3, an example of NFA

Example 4.4: Describe the NFA M4, given in Figure 7, formally.

Solution:
M4 = (Q, Σ, Δ, q0, {q2, q4}), where Q = {q0, q1, q2, q3, q4}, and Σ= {0, 1}. The table of the transitions Δ can be described as follows:

Δ	0	1
q0	{q0, q3}	{q0, q1}
q1	Ø	{q2}
q2	{q2}	{q2}
q3	{q4}	Ø
q4	{q4}	{q4}

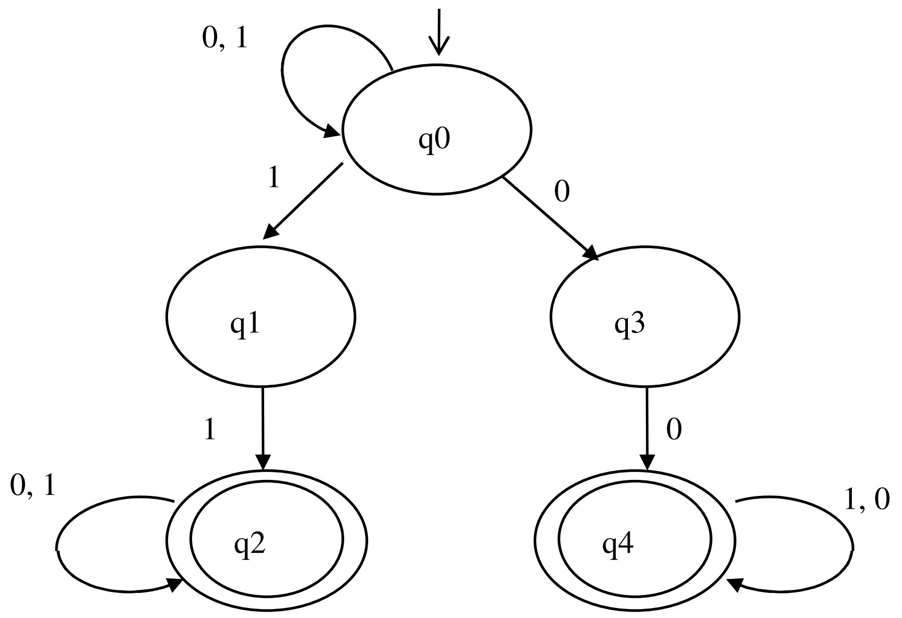

Figure 7: M4, another example of NFA

Example 4.5: Does M4 shown in Figure 7 accept the input 01001?

Sulotion:
We examine the propagation of states of the NFA under the input string 01001. Figure 8 shows that there is a way from q0 to one of the two final states, q4. Therefore, 01001 is accepted by M4.

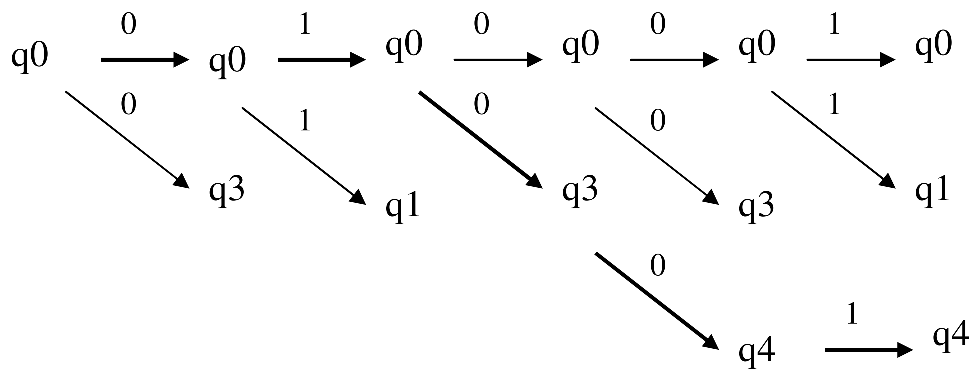

Figure 8: The propagation of states of M4 under the input 01001

[3] **Note:** The extended transition function in NFA, δ*(q, *x*), outputs a set of all states that are reachable from q after reading the string *x*. In the above example, M4, δ*(q0, 01001) = {q0, q1, q4}.

[4] The language of an NFA, M = (Q, Σ, δ, q0, F), can be formally defined as:

$$L(M) = \{w \mid \delta^*(q0, w) \cap F \neq \varnothing\}$$

[5] **Note:** a deterministic finite automaton is just a special type of a nondeterministic finite automaton: In a deterministic finite automaton, it happens that the transition set Δ is in fact a function from Q x Σ to Q. Therefore, every DFA is a NFA by definition.

[6] **Theorem 4.1:** Every nondeterministic finite automaton is equivalent to some deterministic finite automaton. Thus, every NFA has a DFA that recognizes the same language.

[7] **Theorem 4.2:** The languages recognized by NFA are regular. (This follows from Theorem 3.1 and Theorem 4.1)

Exercises on Section 4:

1. Let $\Sigma = \{a, b\}$. Using as few states as possible, draw a NFA that recognizes:
   a. $\varnothing$
   b. all strings that start with a and end with bb

2. Let $\Sigma = \{a, b\}$. Draw a NFA that recognizes
   a. (aa + ab + ba)*
   b. all strings of even length that start with a
   c. all strings that contain the substring abba
   d. all strings that do not contain the substring abba

3. Write a regular expression equivalent to the following NFA:
   a. M3 (shown in Figure 6)
   b. M4 (shown in Figure 7)

4. Consider the NFA M3. Compute: δ(q0, 0), δ(q0, 1), δ(q1, 2), and δ^*(q1, 012)

5. Consider the NFA M4. Compute: δ^*(q0, 1101)